

第3章 变量和控制结构

3.1 语句、复合结构和顺序程序

3.2 变量——概念、定义和使用

3.3 数据输入

3.4 关系表达式与逻辑表达式

3.5 语句与控制结构

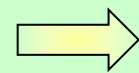
3.6 条件语句

3.7 循环语句

3.8 程序动态除错方法（一）

3.5 语句与控制结构

- 基本语句包括赋值、函数调用等，完成基本操作。
- 复杂计算要通过许多基本操作完成，操作必须按照一定前后**顺序**进行。
- 为描述基本操作的执行过程（**流程**），语言必须提供描述执行流程的机制（**控制结构**）
- 硬件层次的流程控制：**顺序**和**转移**指令
- 低级流程控制的缺点：随意性/程序难以理解
- 程序设计实践总结出三种基本流程模式



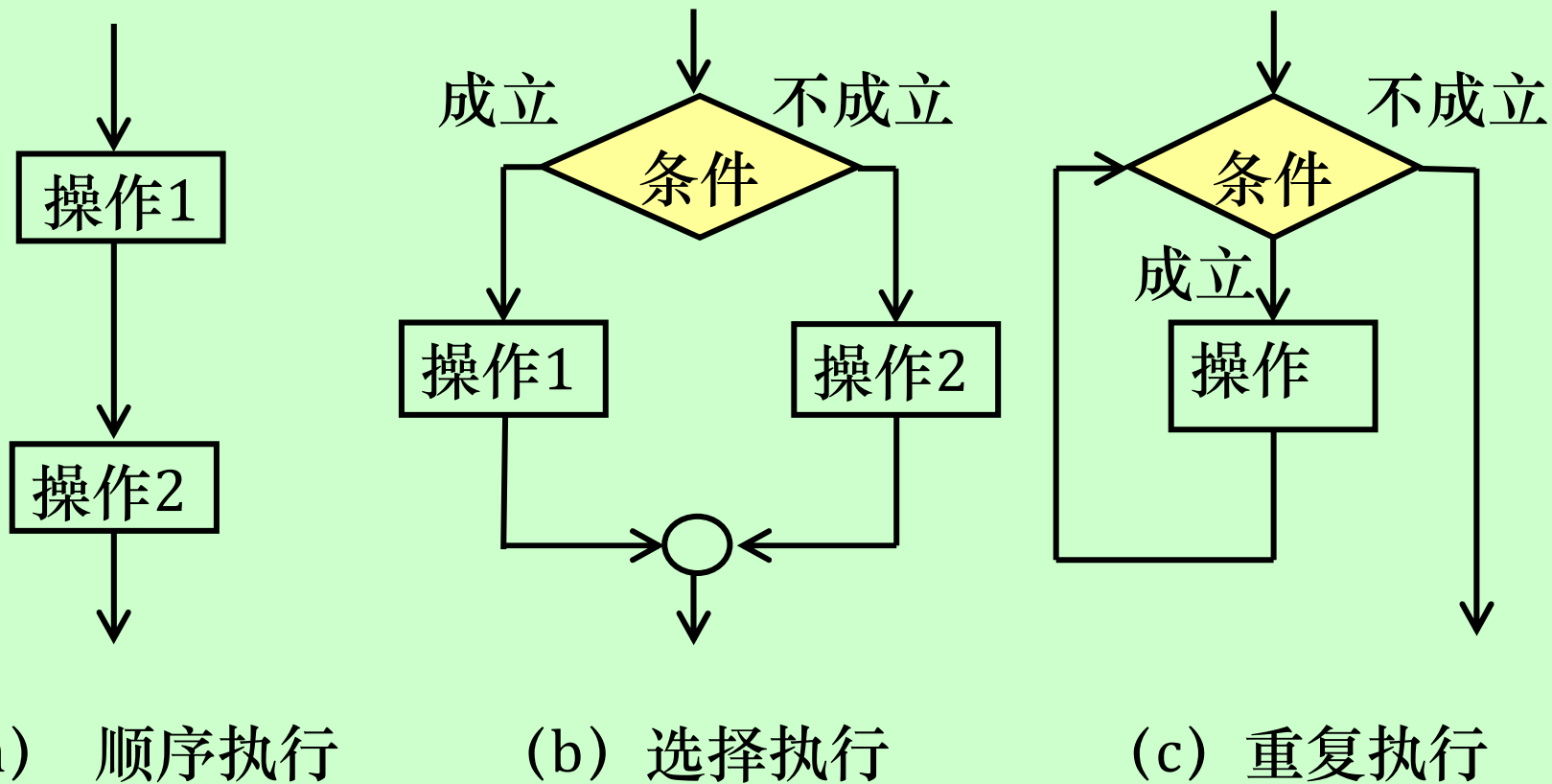


图3-2 程序控制流程的三种基本模式

这里给出了选择和重复的典型情况，还有其他情况。

特点：只有一个开始点/一个结束点。整体可作为抽象操作嵌入各种模式中，形成更复杂的流程。具有层次性，易分解，意义较易把握。

这几个流程模式称为结构化的流程模式。

已证明，这三种模式对写任何程序都够了。

C/C++ 提供了很丰富的控制机制，包括对应上面各种模式的结构化控制结构。

控制结构也被看作语句，也称控制语句。可写在任何能写语句的地方。

已讨论过的复合结构是一种控制结构，实现顺序执行。

第 3 章 变量和控制结构

3.1 语句、复合结构和顺序程序

3.2 变量——概念、定义和使用

3.3 数据输入

3.4 关系表达式与逻辑表达式

3.5 语句与控制结构

3.6 条件语句

3.7 循环语句

3.8 程序动态除错方法（一）

3.6.1 条件语句：if 语句

根据条件判断决定操作是否执行/选择执行。

形式①: **if (条件) 语句**

形式②: **if (条件) 语句1 else 语句2**



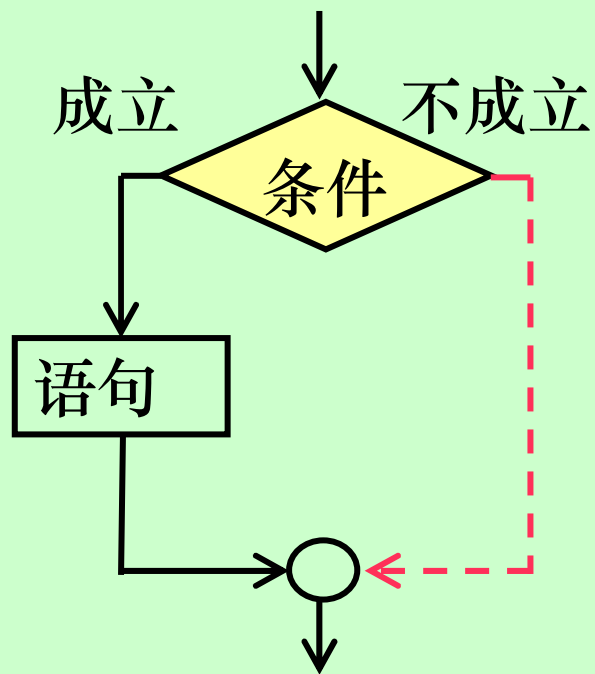
表达式，其值作为控制执行的逻辑值。

语义：

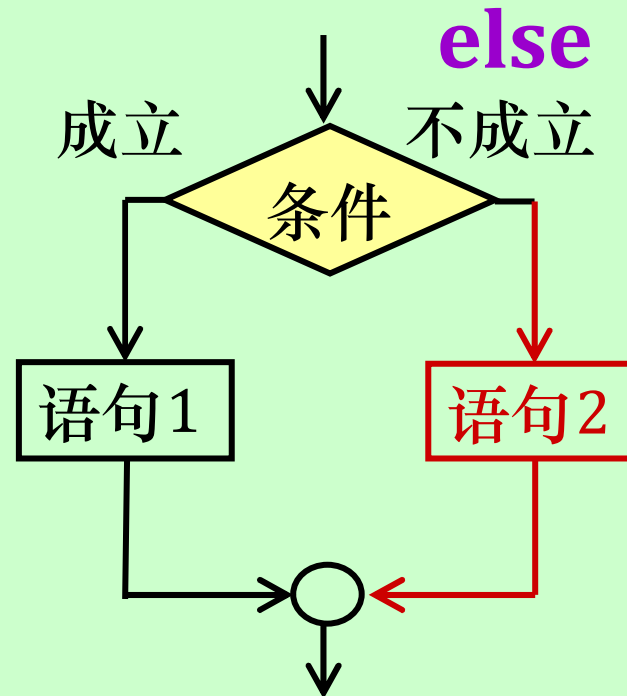
①求值**条件**，条件成立时执行**语句**；

②求值**条件**，条件成立时执行**语句1**；否则执行**语句2**

if (条件) 语句



if (条件) 语句1 else 语句2

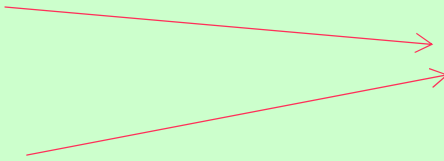


语句、语句1和 语句2 可以是单条语句，也可以是复合结构，还可以是条件语句。

注意概念名词：if条件语句与?: 条件表达式

【例3-10】对于从键盘输入的学生的成绩（0~100），根据其值是否大于等于 60 分而评定一个等级值（'A'或'C'），最后输出等级值。

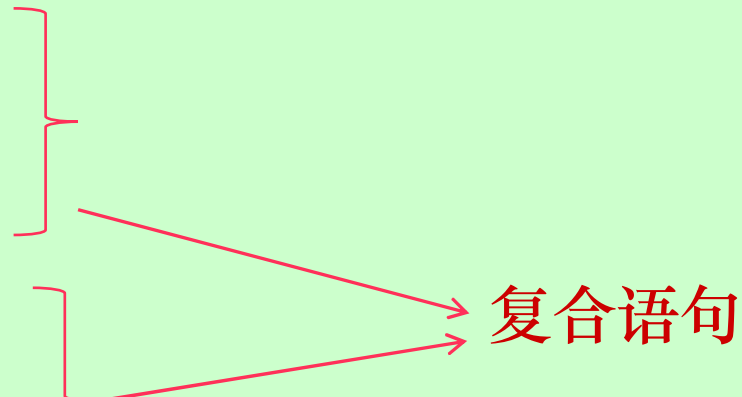
```
int main() {  
    int score;  
    char rank;  
  
    cout << "please input score: ";  
    cin >> score;  
    if (score >= 60)  
        rank = 'A';  
    else  
        rank = 'C';  
    cout << rank << endl;  
    return 0;  
}
```



单条语句

上例稍做修改：在评定一个等级值之后，还要输出“及格”或“不及格”的文字说明。那么，在 if 结构中就要相应地**添加花括号，写成复合语句**。

```
if (score >= 60) {  
    rank = 'A';  
    cout << "及格\t";  
} else {  
    rank = 'C';  
    cout << "不及格\t";  
}
```



复合语句

【例3-11】对于输入的两个整数 a 和 b，如果 a 大于 b，则交换它们的值，最后输出这两个值。

使用一个 if 语句就可以实现题目的要求：

```
int main() {  
    int a, b;  
    cout << "please input a, b : ";  
    cin >> a >> b;  
    if (a>b) {  
        int t = a;  
        a = b;  
        b = t;  
    }  
    cout << "now, a=" << a << ", b=" << b << endl;  
    return 0;  
}
```

形式①的条件语句。

使用了临时变量 t 辅助实现互换。

if (条件) 语句

if (条件) 语句1 else 语句2

3.6.2 if 语句的嵌套

在 if 结构的语句、语句1、语句2可以是任何语句，包括又出现 if 语句。出现这种情况，就称为“if 语句的嵌套”。

【例3-12】对求解一元二次方程 $ax^2 + bx + c = 0$ 的问题，请编写一个程序，使它能根据判别式 $b^2 - 4ac$ 的值来判断方程的实根情况，并且计算出实根的值。

根据数学知识，求二次方程的实根，首先要求出方程判别式的值，根据判别式可以区分出三种情况：该方程有两个实根，有一个重根，或者没有实根。

根据题目要求写出主函数如下，其主要部分是一个嵌套的条件语句。

```
int main() {
    double a, b, c;
    cout << "请输入一元二次方程的三个系数a, b, c: ";
    cin >> a >> b >> c;
    double tmp, d = b*b - 4*a*c;
    cout << "b*b - 4*a*c = " << d << endl;
    if (d > 0) {
        tmp = sqrt(d);
        cout << "Two real roots: " << (-b + tmp) / 2 / a
            << ", " << (-b - tmp) / 2 / a << endl;
    } else if (d == 0)
        cout << "One real root: " << -b / 2 / a << endl;
    else
        cout << "No real root\n";
    return 0;
}
```

- 所写的程序能不能完成我们设想工作呢？
- 除了认真分析问题，安排好计算的步骤并写出代码，反复检查确认所写代码无误外，我们还需要用一些实例仔细检查程序运行的情况。
- 在编写好程序的代码并成功编译之后，就应当用一些实际例子做试验，检查程序输出的情况。

- 程序测试（testing），就是在完成了一个程序或程序的一个部分后，通过一些试验性运行，并仔细检查运行效果，设法确认该程序或部分确实能完成了所期望的工作。也可以反过来说：测试就是设法用一些特别选出的数据去挖掘出程序里的错误，直至无法发现更多错误为止。
- 测试中需要考虑的基本问题就是在运行程序时给它提供什么样的数据，才可能最大限度地将程序中的缺陷和错误挖掘出来。

- 对于我们自己编写的程序，可以根据程序的内部结构和由此而产生的执行流程，而设法选择数据，使程序在试验性运行中能通过“所有”可能出现的执行流程（这也称为“白箱测试”）。如果通过每种执行流程的计算都能给出正确结果，那么这个程序的正确性就比较有保证了。
- 顺序执行的复合语句只有一条执行流，从其中的第一个语句开始，到最后一个语句结束。
- 条件结构“if(条件) 语句”有两条可能的执行流：当条件成立时就执行语句，条件不成立时就不执行语句；“if(条件) 语句1 else 语句2”也有两条执行流：当条件成立时执行语句1，当条件不成立时执行语句2。如果是嵌套的条件语句，则可能产生更多条执行流。

- 因此，如果被程序包含条件语句，测试时就应该提供多种测试数据，检验确认程序在每种执行流程的情况下都能正确完成工作。
- 上面程序有三条可能的执行流（有两个实根、只有一个实根和无实根），因此我们应该根据数学知识，在多次运行程序时分别输入不同的数据（例如“2 5 2”、“1 2 1”和“1 -4 5”），使它能分别通过三条执行流。而且还要仔细检查程序对不同的数据的输出结果，仔细检查这些输出结果是否符合数学结论（通过人工计算进行验证，也可以另写程序来验证）。

if 语句嵌套问题：if 有两种形式，嵌套可能出问题。
问题在条件后直接出现条件语句时。例：

```
if (x > 0)
    if (y > 1) z = 1;
else z = 2; /* 属于哪个 if? */
```

规定：else部分属于前面最近的无对应 else 的 if 语句。
上例的形式易引起误解。

上例实际含义：

```
if (x > 0) {
    if (y > 1)
        z = 1;
    else z = 2;
}
```

要使 else 部分属于外层 if，
可加花括号改变其含义：

```
if (x > 0) {
    if (y > 1)
        z = 1;
}
else z = 2;
```

建议书书写格式

(1) if (XXXXXX)

单条语句

else

单条语句

(2) 当“语句”为复合语句时，常用写法有两种：

```
if (XXXXXX) {  
    XXXXXXXXXXXXXXXX  
    XXXXXXXXXXXXXXXX  
} else {  
    XXXXXXXXXXXXXXXX  
    XXXXXXXXXXXXXXXX  
}
```

上面为推荐格式！

```
if (XXXXXX)  
{  
    XXXXXXXXXXXXXXXX  
    XXXXXXXXXXXXXXXX  
}  
else  
{  
    XXXXXXXXXXXXXXXX  
    XXXXXXXXXXXXXXXX  
}
```

从语义上来说，C/C++是自由语言，随便怎么写都可以。

但是，从实践上来说，我们必须遵循严格的排版缩进格式，以便让排版缩进格式能明显地表示出程序的逻辑结构，从而方便自己能准确无误地把握程序的逻辑。

C/C++ 编程者应该把握的基本的工作态度是：“虽然完成同一件事情的方式很多，但是我应该选取对我最有利的那一种。”

老师会对源代码的编排格式有非常严格的要求。如果不按规定格式来写，自己容易出错；老师可能缺乏耐心来指导。

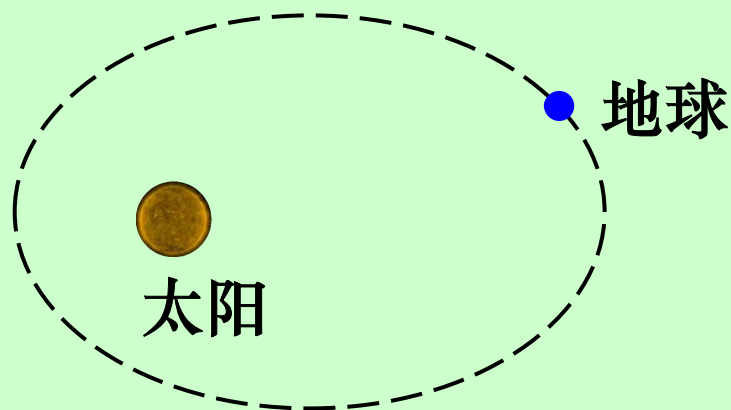
3.6.3 if 语句的优化

if (条件) 语句

if (条件) 语句1 else 语句2

在实际编程中，常常需要对多种条件进行判断并作相应的处理，这时通常有必要对条件进行分析思考，以便对条件语句进行优化，可以使程序写得更简洁。

【例3-13】输入一个年份，判断是否闰年。

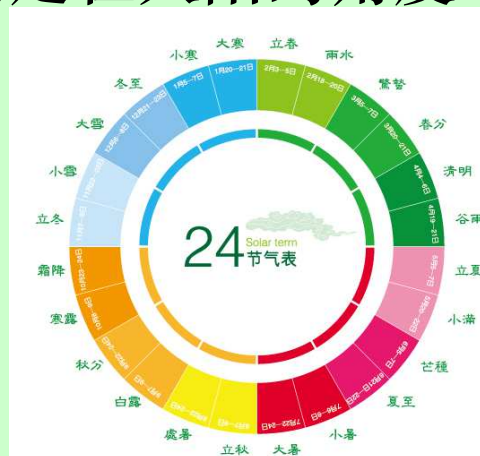


公历闰年小知识

- 地球自转一周为一日，人类生活习惯于以日为单位。
- 地球绕太阳公转一周叫做一回归年，严格的天文观测表明，一回归年长365.2422日（365日5时48分46秒）。
- 为了既使人类能方便地以日为单位生活，又能使长期的历法不产生混乱，现行公历（由教皇格列高利十三世于1582年颁行）规定有平年和闰年。计算方法如下：
 - ◆ 平年一年有365日，比回归年短0.2422日，四年共短0.9688日；
 - ◆ 每四年增加一日，这一年有366日，就是闰年。
 - ◆ 但四年增加一日比四个回归年又多0.0312日，400年后将多3.12日，故在400年中少设3个闰年，也就是在400年中只设97个闰年，这样公历年的平均长度与回归年就非常近似了（每400年多出0.12日，400*8年才多出1日）。
 - ◆ 由此：年份是整百数的必须是400的倍数才是闰年，例如2000年是闰年，而1900年、2100年就不是闰年。

顺便说一下**中国农历**（科普）

- 月球绕地球一周的真实时间是27.32天（天文月）。不过由于地球本身也在绕太阳运行，因此月亮绕到和地球与太阳在一个角度的时间为29.54天。（公历的“月”和“星期”跟天文月毫无关系）
- 中国农历是阴阳历，参照月亮的盈亏，跟天文月和天文年同步。**农历月**的天数是一个变数，有时是29天，有时是30天。农历每月的第一天是月亮全黑的日子。
- **农历年**由24个节气来确定，节气则由太阳的角度来确定。农历的第一个节气叫雨水，定在太阳的角度为330度的日子。其余的23个节气分别定在太阳的角度每变化15度的日子。



初步分析是否为闰年可以按照如下逻辑判断：（1）年份不能被100整除的，如果能被4整除，就是闰年；（2）年份能被100整除的，被400整除才是闰年。

```
int main() {  
    int year, leapyear;  
    cout << "please input a year: ";  
    cin >> year;  
    if (year % 100 != 0)  
        if (year % 4 == 0) //不能被100整除，且能被4整除  
            leapyear = 1;  
        else  
            leapyear = 0;  
    else if (year % 400 == 0) //能被100整除，且能被400整除  
        leapyear = 1;  
    else  
        leapyear = 0;  
    if (leapyear == 1) cout << year << " 是闰年" << endl;  
    else cout << year << " 不是闰年" << endl;  
    return 0;  
}
```

仔细分析，可以改进为满足以下两个条件之一：（1）能被4整除但不能被100整除的都是闰年；（2）能被400整除的年份都是闰年。判断语句可以改进为：

```
if ((year % 100 != 0 && year % 4 == 0) || year % 400 == 0)
    leapyear = 1;
else
    leapyear = 0;
```

如果把变量 leapyear 设置默认值为 0，还可以简化：

```
int leapyear = 0;
if ((year % 100 != 0 && year % 4 == 0) || year % 400 == 0)
    leapyear = 1;
```

或者直接进行逻辑赋值：

```
leapyear = ((year%100 != 0 && year%4 == 0 )
            || year%400 == 0);
```


逻辑判断之后，输出时如果利用条件表达式，可以更简单：

```
cout << year << (leapyear? " 是闰年" : " 不是闰年") << endl;
```

甚至可以把闰年判断和结果输出合并，省略变量 leapyear 。
这样就能把 main 函数中的核心语句改写为下面形式：

```
cout << year << (((year%100 != 0 && year%4 == 0) ||  
    year%400 == 0) ? " 是闰年" : " 不是闰年") << endl;
```

可见，某些情况下，可以用条件表达式代替条件语句。

请特别注意条件语句与条件表达式的不同。**条件表达式**根据给定条件决定求值方式，其基本目的是**算出一个值**。

条件语句的作用是根据条件的成立与否决定做什么，执行什么语句。**条件语句并没有值的概念**。

当然，许多情况下两种结构都可以用，这时就应该从程序的简洁清晰等方面考虑和选择。也有些情况下两种写法在各方面的差异都不大，这时可以根据自己的喜好选择。

由上题的多种写法可见，写程序时**有必要仔细分析条件语句的逻辑判断条件，这样做可能简化条件语句。**

同时，还可以考虑整个程序的流程优化，设法使程序更简洁。

对源程序的基本要求是“**描述简洁，逻辑清晰易理解**”。

这些例子是帮助读者开拓思路。在考虑具体问题时，还是需要具体情况选择合理的描述方式。

【例3-14】请写出程序，对用户输入的三个整数a、b和c，输出其中最大值。

要输出三个数中的最大值，需要比较这三个数。

```
int main() {  
    int a, b, c;  
    cout << "请输入三个整数:";  
    cin >> a >> b >> c;  
    cout << "最大的数是: ";  
    if (a >= b)  
        if (a >= c) cout << a;  
        else cout << c;  
    else // a < b  
        if (b >= c) cout << b;  
        else cout << c;  
    return 0;  
}
```

这个程序采用的方法比较复杂。换种想法，引入一个临时变量 mx，记录已完成的比较确定的（临时）最大值。

改写程序如下：

```
int main(){
    int a, b, c, mx = INT_MIN;
    cout << "请输入三个整数: ";
    cin >> a >> b >> c;
    if (a >= b)
        mx = a;
    else
        mx = b;
    if (mx < c)
        mx = c;
    cout << "最大的数是: " << mx;
    return 0;
}
```

用变量 mx 记录已知的最大值，定义时将其初始化为最小的 int 值。

优点：再多处理一个数，只需要加一个 if 语句，程序更容易修改，长度增加的也不多。

如果用条件表达式，都可以写得更简洁：

```
mx = (a >= b ? a : b);
```

```
cout << (mx >= c ? mx : c);
```

3.6.4 使用 if 语句的技术

if (条件) 语句

if (条件) 语句1 else 语句2

括号里面需要写一个条件。有读者可能想当然地认为，这里必须写关系表达式或逻辑表达式。

实际上，条件可以是任何基本类型的表达式（常数或变量也是表达式），其值将被 if 语句当作逻辑值使用。

算术表达式：a + b

关系表达式：a > b

逻辑表达式：a > b && b < c

条件表达式：a > b ? 1 : 2

(1) 如果一个if结构中需要用**整型变量 k 的值不等于零**作为执行条件，直观的写法是：

```
if (k != 0) { ... }    // ... 表示其它语句
```

然而，注意到 **k 的值本身就可以作逻辑值来使用**，采用下面写法的效果完全一样：

```
if (k) { ... }
```

if 语句也允许用常数作为条件：

```
if (1) { ... }
```

由于 1 表示真（条件成立），这是一个条件始终为真的 if 语句，虽然符合语法，没什么实际价值。

(2) 用 **k 的值是否等于零** 作为条件，直观的写法是：

```
if ( k == 0 ) { ... }
```

利用 **k 的值本身也可以作为逻辑值**的情况，可写成：

```
if (!k) { ... }
```

(3) 用 k 是否等于某个固定值（例如10）作为条件：

```
if (k == 10) { ... }
```

也可以写成

```
if (k - 10 == 0) { ... }
```

或者

```
if (k - 10) { ... }
```

使用 if 语句时的常见错误

- 把 `==` 误写成 `=` :

```
if (k = 10) { ... } //错误示例
```

可以故意写成这样以避免出错:

```
if (10 == k) { ... } //
```

- 在 if 结构的条件之后多写了一个分号:

```
if (k == 10);  
    cout << "k 等于 10";
```

=

```
if (k == 10)  
    ;  
    cout << "k 等于 10";
```


3.6.5 开关语句

开关语句是一种多分支结构，用于实现多个分支中的选择执行。开关语句的一般形式是：

```
switch (整型表达式) {  
    case 整型常量表达式: 语句序列; break;  
    case 整型常量表达式: 语句序列; break;  
    ....  
    default: 语句序列 ; break;  
}
```

语句头部的整型表达式用于选择分支。各个case关键词后面的整型常量表达式（下面称其为case表达式）必须在编译时就能确定值，常用整型或字符型的文字量。这些“case 整型常量表达式:”当作标号看待，不同case表达式的值必须互不相同。default开始的段可以没有。各个case和default之后的语句序列可以包含任意多个语句，也允许不包含语句（可以是空序列）。

```
switch (整型表达式) {  
    case 整型常量表达式: 语句序列; break;  
    case 整型常量表达式: 语句序列; break;  
    ....  
    default: 语句序列; break;  
}
```

开关语句的执行过程如下:

- 首先求出语句头部的整型表达式的值，然后用这个值与各个case表达式的值比较。
- 如果遇到相等的值，就进入那个case之后的语句序列开始执行；遇到 break 就结束执行。
- 如果找不到匹配的值，而这一开关语句有default部分，就执行default部分的语句序列；
- 如果找不到匹配的case表达式，又没有default部分，整个开关语句的执行结束。

系统执行完一个case的语句序列后，如果这个switch语句还没结束，就接着执行下一个（case之后的）语句序列。

经常希望只执行switch语句中一个分支的语句序列，该序列执行完成后就结束这个switch语句。为此，需要在各个分支的语句序列最后加一个break语句。一旦执行到break语句，这个switch语句就结束。为了代码的规范性，人们也习惯在default的语句序列最后也写一个break语句。

break语句在形式上就是一个关键字（加结束的分号）：

`break;`

它只能用在switch语句以及在后面将要介绍的循环语句里。用在switch语句里时，break语句的作用是使当前switch语句立刻终止，使程序转到这个switch语句之后继续执行。

```
switch (整型表达式) {  
    case 整型常量表达式: 语句序列; break;  
    case 整型常量表达式: 语句序列; break;  
    ....  
    default: 语句序列; break;  
}
```

当case分支的最后没有break语句时，它的语句序列执行完成后程序将进入下一分支的语句序列，这种情况导致一些分支的语句序列被共享。一般认为，除非多个分支都用完全一样的语句序列（也就是说，一些case标号后面没有语句序列，和后面标号使用同一个语句序列），否则不提倡这种代码共享的方式。部分代码共享将导致程序片段间的依赖关系，不利于程序的修改。

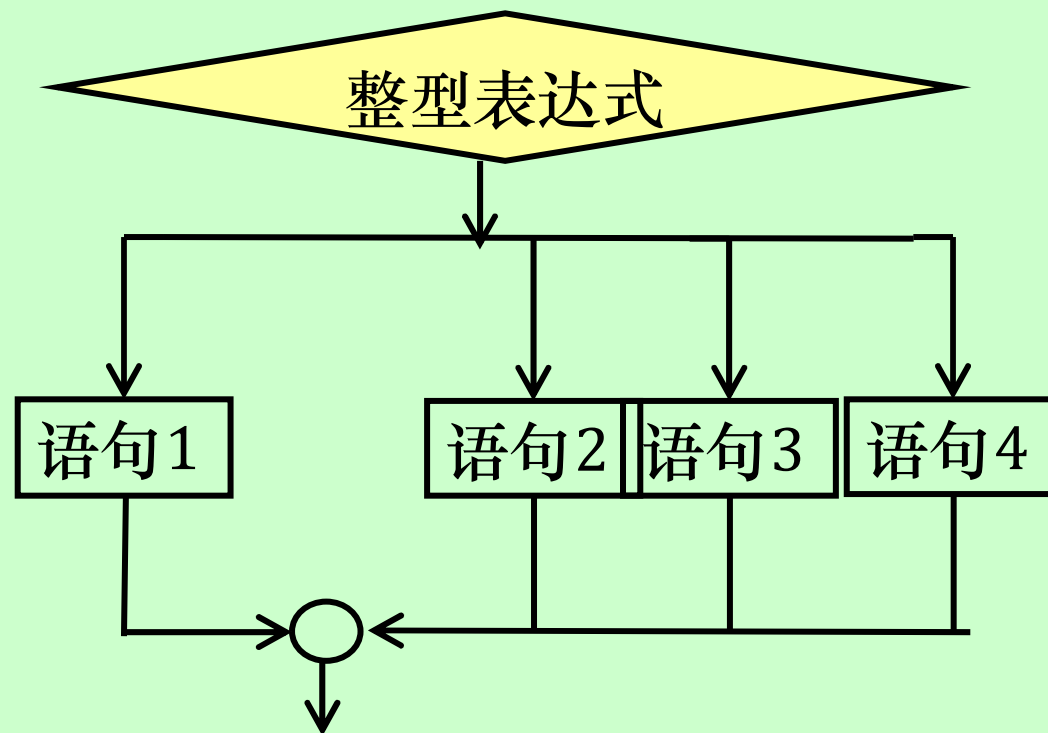
【例3-15】 写一个程序，它输入一个表示学生考试成绩的整数（评分），根据这个成绩给学生赋一个等级，每10分为一个等级，[90, 100] 范围内为“优”，[80-90) 范围内为“良”，[70, 80) 范围内为“中”，[60, 70) 范围内为“及格”，其余为“不及格”。

可以写一串 if 语句完成这一工作，考虑到这里需要区分5种情况，用switch语句也是一种可能性。

由于分级的情况比较规范，每10分为一级（只有100分比较特殊），把每一段归结到一个整数，把成绩值整除10得到的整数可用于在switch结构里选择分支：整除的商值为10和9时共享同一段代码，值为8、7、6时分别处理，其它值为默认处理。

利用switch结构写出主函数如下：

```
int main() {  
    int score, rank;  
    cout << "Input score(0~100): ";  
    cin >> score;  
    rank = score / 10;  
    switch(rank) {  
        case 10:  
        case 9:  
            cout << "优 " << endl; break;  
        case 8:  
            cout << "良 " << endl; break;  
        case 7:  
            cout << "中 " << endl; break;  
        case 6:  
            cout << "及格 " << endl; break;  
        default:  
            cout << "不及格" << endl; break;  
    }  
    return 0;  
}
```



第3章 变量和控制结构

3.1 语句、复合结构和顺序程序

3.2 变量——概念、定义和使用

3.3 数据输入

3.4 关系表达式与逻辑表达式

3.5 语句与控制结构

3.6 条件语句

3.7 循环语句

3.7.1 while 语句

3.7.2 do-while循环结构

3.7.3 for 语句

3.7.4 与循环有关的控制语句

3.7.5 死循环

3.8 程序动态除错方法（一）

3.7.1 循环语句(1): while 语句

循环控制结构实现特定条件下某些操作的重复执行。

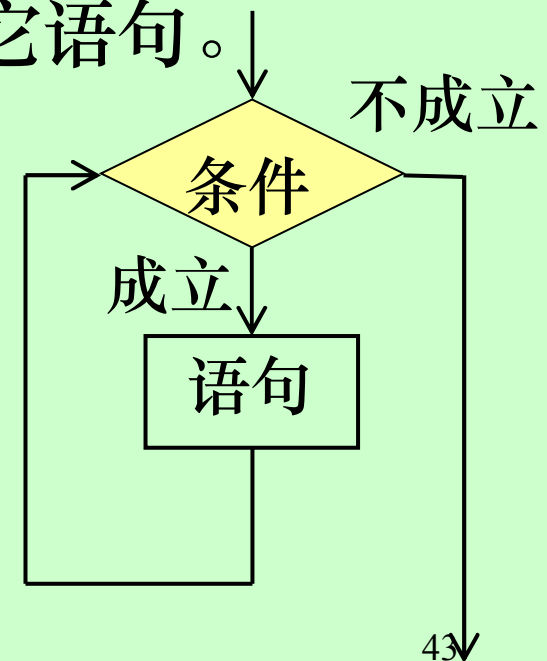
C/C++有多种循环结构，while 语句最简单，形式：

while (条件) 语句

语句为循环体，可以是单条语句，也可是复合语句，还可以是循环（构成多重循环）或其它语句。

语义：

- (1) 求值**条件**表达式，将它的值作为逻辑值
- (2) 若第1步得到值0则循环结束；否则
- (3) 执行作为循环体的**语句**，而后回到 (1)



【例3-16】角的度量单位通常有两种：角度制和弧度制。

在**角度制**中，一周被划分成 360 度（degree，通常缩写为deg），角的大小以度数给出。

弧度制是用弧的长度来度量角的大小，单位弧度定义为圆周上长度等于半径的圆弧与圆心构成的角。角度以弧度给出时，通常不写弧度单位，有时记为 rad 或 R。

弧度制和角度值转换关系为：

$$\text{弧度数} / \pi = \text{角度值} / 180^\circ。$$

写一个程序，要求它从角度值 0 度到 180 度，每隔 5 度为一项，计算并输出一个角度与弧度的对照表。

考虑写循环：

需要对一系列角度求对应的弧度。这些弧度可以按统一的规律一个个算出来。（变化的量）

- 用一个 int 变量保存角度值
- 循环前赋初始值 0，每次循环加 5，直到 180
- 用一个 double 变量保存弧度值。每个角度值转换为弧度值，输出一行，显示当时角度 - 弧度对照

这形成了一套解决问题的方案。

```
#include <iostream>
using namespace std;
```

```
int main () {
    int deg = 0; //定义变量并初始化
    double rad;
    cout << "deg \trad" << endl;
    while (deg <= 180) {
        rad = 3.1415927 * deg / 180; //错误: deg/180*3.1415927
        cout << deg << "\t" << rad << endl;
        deg = deg + 5;
    }
    return 0;
}
```

while (条件) 语句

while (deg <=180) { ... }

【例3-17】 写程序求出数学式 $\sum_{n=1}^{100} n^2$ 的值。

```
int main() {  
    int n = 0, sum = 0;  
    while (n < 100) {  
        n = n + 1;  
        sum = sum + n * n;  
        cout << "n = " << n << " sum = " << sum << endl;  
    }  
    cout << "Result: sum = " << sum << endl;  
    return 0;  
}
```

典型形式：循环前给一些变量初值；循环中修改某些变量。
用包含不断变化的变量的条件控制循环是否结束。

循环体可能多次执行，引起很长计算，甚至无限长。如何分析问题/写出循环很重要（第4章讨论）

```
int main() {  
    int n = 0, sum = 0;  
    while (n < 100) {  
        n = n + 1;  
        sum = sum + n * n;  
        cout << "n = " << n << " sum = " << sum << endl;  
    }  
    cout << "Result: sum = " << sum << endl;  
    return 0;  
}
```

(2) 在每次循环体的执行中，虽然执行的是同样程序片段，但由于参与循环的一些变量的值改变了，实际做的事情就可能不同。在本程序的循环体中，变量n每次增1，变量sum相应地进行一次累加求和计算。（顺便说一下，在变量更新为+1或-1时，人们更喜欢使用增量运算符。在本程序中， $n = n + 1$ 是一个单独的表达式，不涉及到其它变量，所以写成 $n++$ 或 $++n$ 皆可。）

(3) 显然，一个循环结构需要有一个继续条件（不满足此条件时循环终止），它控制着循环的进行过程。在上例中，继续条件是 $n < 100$ ，所以在n的值为99时仍然进入循环体，执行完循环体内的语句之后n的值变为100，此时循环条件变为假，循环终止。也就是说，循环结束时，n的值为100（如果后续还要使用n的值，必须注意这一情况）。

在仔细地考虑了循环的初始值、参与循环的变量变化情况、循环结束条件等因素之后，上面程序中的循环也可以改为如下这样：

```
int n = 1, sum = 0;
while (n <= 100) {
    sum = sum + n * n;
    cout << "n= " << n << " sum= " << sum << endl;
    n++;
}
```

注意不同之处.....

```
int main() {  
    int n = 0, sum = 0;  
    while (n < 100) {  
        n = n + 1;  
        sum = sum + n * n;  
        cout << "n = " << n << " sum = " << sum << endl;  
    }  
    cout << "Result: sum = " << sum << endl;  
    return 0;  
}
```

```
int n = 1, sum = 0;  
while (n <= 100) {  
    sum = sum + n * n;  
    cout << "n= " << n << " sum= " << sum << endl;  
    n++;  
}
```

读者可以从这两种写法中体会到，描述循环时需要细致的逻辑思考。循环语句各个要素的细节都必须精确无误，不能随意。在任何一个细节上出错都会导致最终结果出错。

在循环体中输出变量的中间值，有助于我们在观察变量的变化情况，检查程序的运行情况和结果。

在当前常见的Windows系统下运行时可以发现，当求和上限值为2000时，最终输出结果就不正常了。仔细观察可以看到，实际上是在n的值为从1860变到1861时，sum的值发生了突变，居然从一个正数变成了负数：

n= 1859 sum= 2143222510

n= 1860 sum= 2146682110

n= 1861 sum= -2144821865

n= 1862 sum= -2141354821

发生了溢出错误，所以后续计算的值都是错误的。

因此，如果在程序中写循环进行累加、累乘时，一定要考虑避免出现溢出错误。

如果把变量 sum 的类型改为double，那么可以避免结果溢出变为负数的明显错误，但是累加和的结果是用浮点数表示，并不能得到精确的整数值，而且当 n 太大时最终也会出现溢出。

3.7.2 do-while循环结构

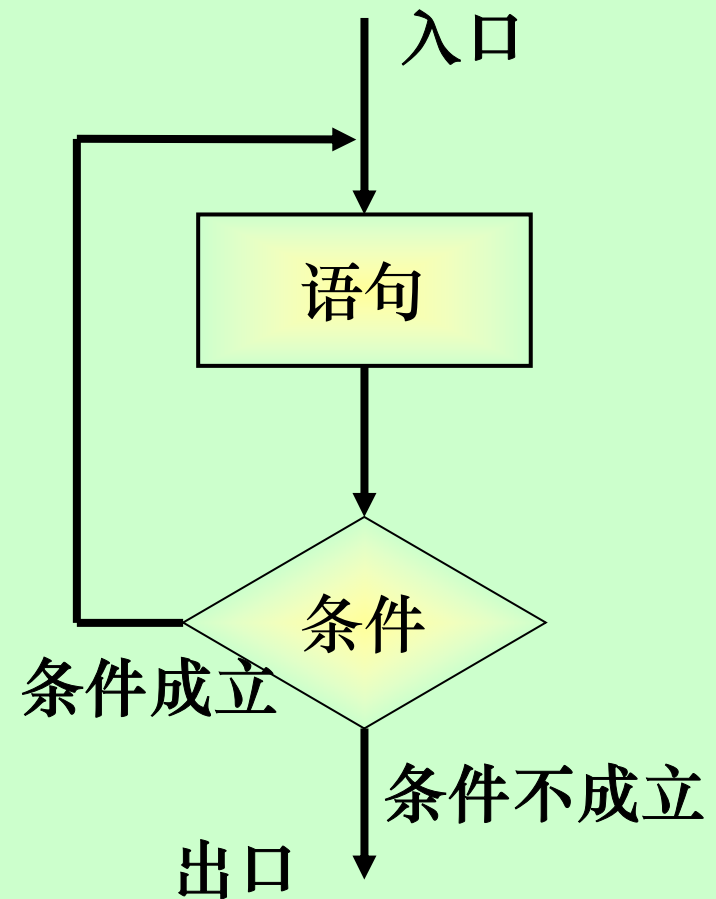
语法形式：

do 语句 while(条件);

执行过程如图。

与 while 的差异在于**判断在后**，至少执行 语句 一次。

do-while 使用较少。



do-while结构的执行流程

用do-while语句重写的求 $\sum_{n=1}^{100} n^2$ 的代码段:

```
int main() {  
    int n=0, sum = 0;  
    do {  
        n++;  
        sum = sum + n * n;  
        cout << "n= " << n << " sum= " << sum << endl;  
    } while (n < 100);  
    cout << "Result: sum =" << sum << endl;  
}
```

do { ... } while(条件);

上机练习内容：

3.6 条件语句：例题3-10 ~ 3-15

- 编程练习：3-6, 3-7, 3-8, 3-9

3.7 循环语句：例题3-16, 3-17

- 编程练习（只练习 while 和 do-while）：3-10, 3-11, 3-13
- 文件名要求按照以前的规定来命名。
- 上交文件：xxxx-xxx-prog3-9, xxxx-xxx-prog3-10
- 具体见群内通知

3.7.3 循环语句(3): for 语句

最常见的循环模式：(1)变量赋初值；(2)检查循环条件；(3) 成立时执行循环体；(4) 更新变量并继续。

求 $\sum_{n=1}^{100} n^2$ 的代码段：

```
int n = 0, sum = 0;
while (n < 100) {
    n = n + 1;
    sum = sum + n * n;
    cout << "n = " << n
    << " sum = " << sum << endl;
}
```

```
int n = 0, sum = 0;
do {
    n++;
    sum = sum + n * n;
    cout << "n= " << n << "
sum= " << sum << endl;
} while (n < 100);
```

3.7.3 循环语句(3): for 语句

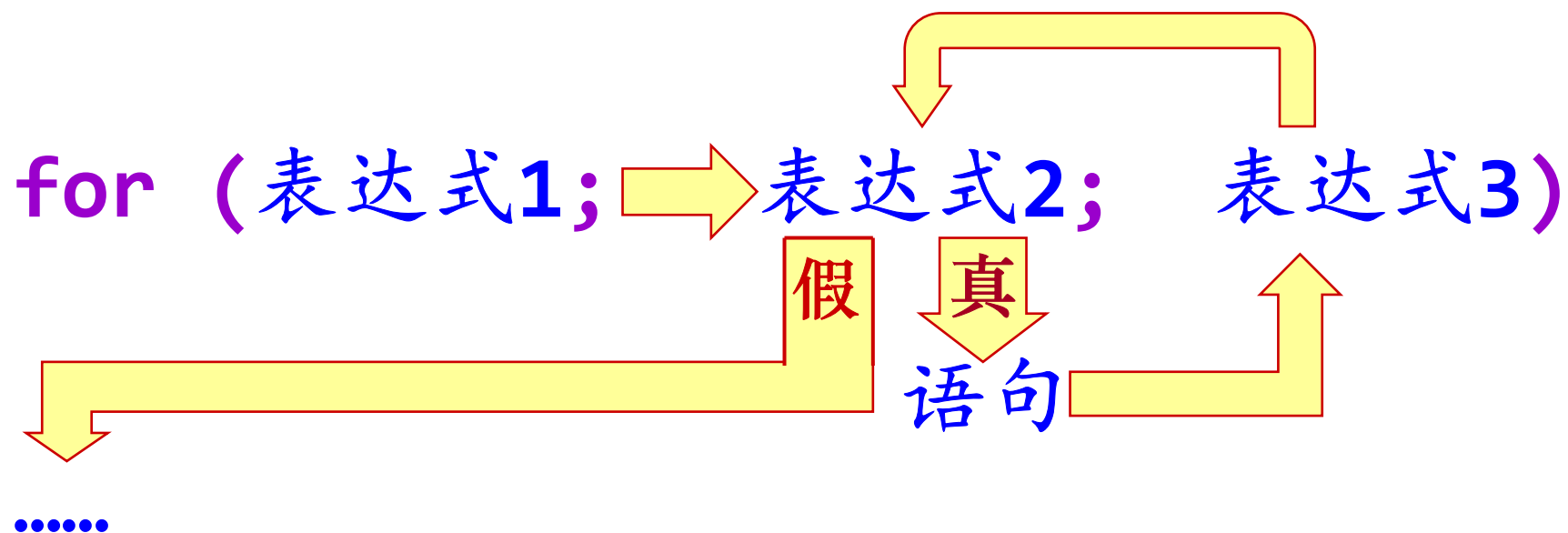
最常见的循环模式：(1)变量赋初值；(2)检查循环条件；(3) 成立时执行循环体；(4) 更新变量并继续。

for 结构是这种常见模式的规范化。形式：

for (表达式1; 表达式2; 表达式3) 语句

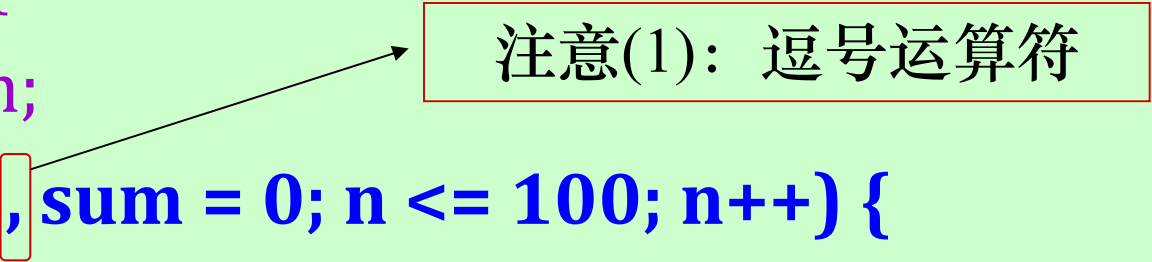
执行（语义）：

- (1) 求值**表达式1**（只做一次），用于**设变量初值**
- (2) 求值**表达式2**，值为 0 时循环结束（**循环条件**）
- (3) 执行**语句**
- (4) 求值**表达式3**，通常用于**循环变量更新**
- (5) 转到（2）



用 for 语句重写的求 $\sum_{n=1}^{100} n^2$ 的代码段:

```
int main() {  
    int n, sum;  
    for(n = 1, sum = 0; n <= 100; n++) {  
        sum = sum + n * n;  
        cout << "n= " << n << " sum= " << sum << endl;  
    }  
    cout << "sum= " << sum << endl;  
    return 0;  
}
```



注意(1): 逗号运算符

(2) 循环体内如果只有一条简单语句，可以不用花括号：

```
for(n = 1, sum = 0; n <= 100; n++)
```

```
    sum = sum + n * n;
```

(3) 当循环体内只有一条简单语句时，甚至可以把它合并到for结构的语句3中，而循环体中写一个空语句。例如写成：

```
for (sum = 0, n= 0; n <= 100; n++, sum += n * n)
```

```
    ; //空循环
```

注意，for 语句头部的圆括号后不要多写分号：

```
for(n = 1, sum = 0; n <= 100; n++);
```

```
    sum = sum + n * n;
```

以上代码执行情况如何？



(4) 也可以在定义变量时初始化，让**表达式1**为空。例如，程序中的主要部分可以写成：

```
int n = 1, sum = 0;  
for(; n <= 100; n++) {  
    sum = sum + n * n;  
    cout << "n= " << n << " sum= " << sum << endl;  
}
```

这种写法也可以正常运行。但把对变量赋初值的功能移到了for结构之外，使for结构的功能就变得不够完整独立了。不建议采用这种写法。

for 头部各表达式都可缺，但必须保留分号。
如果缺少**表达式1** 或 **表达式3**表示无相应动作；
缺**表达式2**表示条件为1（可用其他机制退出循环）

(5) n 和 sum 这两个变量的功能不同。sum 是循环之后将要输出的累加和，可以看作循环的主要结果，而 n 只是在循环过程中使用到的变量，在上例的循环之后就没用了。

人们建议变量的定义应尽可能局部化。为此，for 结构支持表达式1 中定义只在本循环语句中使用的变量。

可以把 n 定义为只在 for 语句内部起作用：

```
int sum = 0;
for (int n = 1; n <= 100; n++) {
    sum = sum + n * n;
    cout << "n= " << n << " sum= " << sum << endl;
}
```

注意，只有在 for 头部括号里的开始处可以出现一个变量说明，这里可以定义一个或几个同类型的变量，它们都只能在这个 for 语句中使用，都必须在这里初始化。

能不能把对sum的定义也写在for语句里面？

答案是：如果把变量 sum的定义移到 for 循环头部，它就是只在这个 for 语句里可以用的变量，在 for 语句之后不再有定义。因此，编译器会报告 for 结构之后的输出语句中的 sum无定义。

对这个问题在后面章节里有详细解释（参见第5章中的“5.1.5 局部变量的作用域和生存期”）。

暂且把这样当作一个规则：**在 for 结构里使用的变量可以在 for 结构的 表达式1 部分定义，这种变量只在 for 结构的头部和循环体里可用。**

而需要在 for 结构之外使用的变量，应该在 for 结构之前定义并初始化。

二、for语句常用技巧

- 既可用“向上循环”，也可用“向下循环”。：

```
int sum = 0;
for (int n = 100; n >= 1; --n) {
    sum = sum + n * n;
    cout << "n= " << n << " sum= " << sum << endl;
}
```

- 循环时可取具有同等间隔的值：

例：求 [13, 315] 间每隔7的各整数的平方根之和。

```
for (sum = 0.0, n = 13; n <= 315; n += 7)
    sum += sqrt(n);
```

- 一般不用浮点数控制循环，尤其是增量为小数或包含小数时。

例：求从0到100每隔0.2的数的平方根之和：

```
double sum, x;
```

```
for (sum=0.0, x=0.2; x<=100.0; x+=0.2)
```

```
    sum += sqrt(x);
```

由于浮点计算误差，不能保证循环500次。

应该写成：

```
int n; double sum;
```

```
for (sum = 0.0, n = 1; n <= 500; ++n)
```

```
    sum += sqrt(0.2*n);
```

三、for语句互相嵌套

各种控制结构都可以互相嵌套，下面是一个利用for循环嵌套完成工作的简单例子。

【例3-19】 输出九九乘法表。

每个被乘数输出一行，从1到9共输出9行。可以用常规形式的for循环完成，其中**用一个取值从1开始到9结束的循环变量m**，既用于控制循环，也作为被乘数。

对每个被乘数，需要产生它与从1到9的乘数的乘积。乘数取值从1到9，用for循环描述特别方便。我们用**另一个取值从1开始到9结束的循环变量n**，同样既用于控制循环也作为乘数。形成了一个两层嵌套的for循环结构。

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main() {
    int m, n;
    cout << "9*9 multiplication table" << endl;
    for (m = 1; m <= 9; m++) {
        for (n = 1; n <= 9; n++)
            cout << m << "*" << n << "=" << setw(2) << m * n << " ";
        cout << endl;
    }
    return 0;
}
```

也可写为: $n \leq m$;

注意:

【例3-20】 两个乒乓球队进行比赛，各出3人。甲队队员为A、B和C，乙队队员为x、y和z。通过抽签决定比赛名单。有人在抽签后（尚未正式公布之前）向队员打听比赛的名单，A说他不与x比赛，C说他不与x和z比赛。请编程找出3对赛手的名单。

这个题目是一个逻辑推理问题，在编程时可以通过固定某一组队员、让另一队员循环轮换并检测是否满足已知条件来解决。在此我们选择固定甲队三个队员的顺序为A、B和C，然后让乙的队员进行循环轮换。假设i是A的对手，j是B的对手，k是C的对手，可以让i、j和k控制进行三层循环。

由于在C/C++程序中可以把字符当作取值较小的整数来用，因此可以把i、j和k直接对三个连续的字符'x'、'y'和'z'取值进行循环。对于这三个变量的每一种取值情况进行检测时，不仅要求i、j和k互不相等，同时还要满足条件 $i \neq 'x'$ 、 $k \neq 'x'$ 和 $k \neq 'z'$ 。

```
int main() {  
    char i, j, k; //i是A的对手， j是B的对手， k是C的对手  
    for (i = 'x'; i <= 'z'; i++)  
        for (j = 'x'; j <= 'z'; j++)  
            for (k = 'x'; k <= 'z'; k++)  
                if (i != j && i != k && j != k  
                    && i != 'x' && k != 'x' && k != 'z')  
                    cout << "A -- " << i << "\nB -- " << j  
                        << "\nC -- " << k << endl;  
    return 0;  
}
```

- 上面介绍了全部三种循环结构，程序中需要描述重复性计算时，我们可以根据情况选择使用。
- while 和 do-while 循环结构比 for 结构要简单。两者之中，while 循环用得较多。
- for 语句的功能更紧凑、强大，实际覆盖了while的功能。for 循环的结构比较复杂，成分多，但它的设计确实反映了循环的典型特征，也很常用。
- for 语句执行的循环次数 不是事先确定的。循环初始值、循环条件、变量更新操作都影响 for 语句的循环次数
- 各种控制结构都可以相互嵌套使用，写出更复杂的程序，解决更复杂的计算问题。

3.7.4 与循环有关的控制语句

一、利用标志变量或 break 语句退出循环

【例3-21】假定希望写一个程序，它从键盘接受一个正整数，判断其是否为质数并输出结果。

判断一个数（例如 n ）是否质数，最直接而简单的方法，是设法确定它有无真因子。

整数 k 整除 n 可以用条件 $(n \% k == 0)$ 描述，如果 $k < n$ 而且 k 不是1，那它就是 n 的真因子。

一种检查质数的简单方法：令变量 k 由 2 开始递增取值，一个个试除 n ，直至完成判断。

整个工作可以通过一个循环完成，如果循环中找到 n 的真因子，就可以确定 n 不是质数；如果直到循环结束也没找到真因子， n 就是质数。

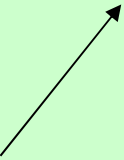
循环结束后需要知道是否找到过真因子。一种可行方法是引进一个bool类型的变量 `found`。false表示未发现真因子，发现真因子给它赋值true。循环初始时应该置为 false。

```
int main() {  
    int n, k;  
    bool found;  
    cout << "please input a positive integer: ";  
    cin >> n;  
    for (found = false, k = 2; k < n; k++) {  
        if (n % k == 0)  
            found = true;  
    }  
    if (found) cout << n << " 不是质数" << endl;  
    else cout << n << " 是质数" << endl;  
    return 0;  
}
```

改为 “`k * k < n;`” 可以
明显减少循环次数

```
int main() {  
    int n, k;  
    bool found;  
    cout << "please input a positive integer: ";  
    cin >> n;  
    for (found = false, k = 2; not found && k * k < n; k++) {  
        if (n % k == 0)  
            found = true;  
    }  
    if (found) cout << n << " 不是质数" << endl;  
    else cout << n << " 是质数" << endl;  
    return 0;  
}
```

found 为真表示发现了
n 的真因子，可以利用
它更早结束循环



break 语句还可以用在循环语句里，其作用就是使当前的（最内层的，因为循环等可能出现嵌套）循环语句立刻终止，使程序从被终止的循环语句之后继续执行下去。

```
int main() {  
    int n, k;  
    cout << "please input a positive integer: ";  
    cin >> n;  
    for (k = 2; k * k <= n; k++)  
        if (n % k == 0)  
            break;  
    cout << n << ((k * k <= n && n % k == 0)?  
        "不是质数": "是质数") << endl;  
    return 0;  
}
```

通常应把break语句放在条件语句控制之下，以便在某些条件成立时立即结束循环。

其它细节:

```
int main() {  
    int n, k;  
    cout << "please input a positive integer: ";  
    cin >> n;  
    for (k = 2; k * k <= n; k++)  
        if (n % k == 0)  
            break;  
    cout << n << ((k * k <= n && n % k == 0)?  
        "不是质数": "是质数") << endl;  
    return 0;  
}
```

for 循环结束有两种可能:

- 1、循环到 $k * k > n$;
- 2、找到 n 的真因子, break。

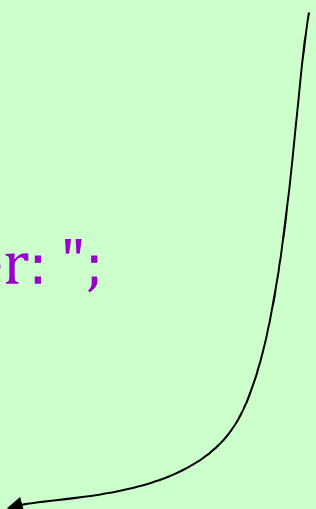
if (n % k == 0)
 break;

单条语句

根据这两个条件判断质数

也可以找到真因子就输出结果，然后直接结束整个程序。

```
int main() {  
    int n, k;  
    cout << "please input a positive integer: ";  
    cin >> n;  
    for (k = 2; k * k <= n; k++)  
        if (n % k == 0) {  
            cout << n << "不是质数" << endl;  
            return 0;  
        }  
    cout << n << "是质数" << endl;  
    return 0;  
}
```



二、用continue语句改变循环过程

continue 语句 的形式:

continue;

只能用在循环里，使当前循环体的一次执行结束，进入下次循环。

while/do-while的随后动作是条件判断；

for的随后动作是变量更新。

```
while (...) {
```

```
... ..
```

```
... ..
```

```
... break;
```

```
... ..
```

```
... ..
```

```
... continue;
```

```
... ..
```

```
... ..
```

```
}
```

break 和 continue 语句
引起的控制转移

【例3-22】 写一个程序，输出0~100之间所有不能被3整除的整数。

用一个循环输出0~100，题目要求我们跳过所有能被3整除的数。应通过一个条件语句实现。利用条件语句和continue语句完成工作：

```
int main() {  
    int k;  
    for (k = 0; k <= 100; k++) {  
        if (k % 3 == 0) // k 能被3整除，跳过  
            continue;  
        cout << k << endl;  
    }  
    return 0;  
}
```

也可改用 if 语句：

```
for (k = 0; k <= 100; k++)  
    if (k % 3)  
        cout << k << endl;
```

三、goto语句/转移语句/转跳语句

goto 语句与**标号**配合，实现函数体内的任意控制转移。是最老的控制语句。**现在已很少用。**

标号可写在任何语句前面作为goto的目标。形式是(**标号名**是标识符)：

标号名：

goto语句的形式：

goto 标号名；

作用（语义）：使控制转到**标号**处继续执行。

break、continue是受限的goto，实现固定方式的控制转移。循环和分支也是goto的包装。

无节制地用goto写程序，费解，常带有难发现的错误。

1968年Dijkstra撰文“goto是有害的”。六年大辩论的结果是结构程序设计革命，语言都引进“标准”控制结构，教育和实践中提倡结构化程序设计。

对goto的认识：**不用或尽量少用**。

大部分goto实际上是为构造条件或循环：

1) 向前转跳

label:

....

... goto label;

2) 向后转跳

... goto label;

....

label:

用循环或条件重写的程序更清晰易读，不容易有错。

随便使用goto是**不良编程习惯**。

不合理的goto表明对问题欠分析，没做好流程分解，函数抽象等，写的是不成熟的程序。

3.7.5 死循环

某个循环在执行时无穷无尽地重复而永不结束：“死循环”。会导致程序无限执行下去，不能正常地终止运行。

死循环是程序运行中发生的情况，有些死循环可以在代码中明显看到。例如：

```
while(1)
    cout << "*";
```

或

```
for (i = 0; ; )
    ;
```

在实践中，可能由于循环结构的结束条件写得不正确，或者循环体里的代码有错，或者有些情况考虑不周，使得程序（在某些情况下）无法达到循环的结束条件，导致程序运行中出现了死循环。

```
int n = 0, sum = 0;
while (n < 100) { //忘记在循环体中更新变量n的值
    sum += n;
}
```

```
int n, sum = 0;
while (n < 100)
    sum += n;
    n++;           //这句在循环之外
```

```
for (int n = 0; n < 100; n++)
    if(n = 50) //打字错误
        cout << "n equals 50! ";
    else
        cout << " " << endl;
```

要注意避免出现
类似错误！

在实际中，要判断自己的程序确实出现了死循环，也不是很简单的事情。有些源程序写有“while(1){...}”的形式，似乎看上去像是死循环，但如果里面具有能正常退出循环的语句，则不是死循环。有些程序需要运行很长时间，可能也会让读者误以为死循环。下面是一个这样的例子：

```
int main() {  
    int i = 0;  
    while(1) {  
        cout << i << " ";  
        i++;  
        if (i == 1000000)  
            break;  
    }  
    return 0;  
}
```


还有一种可笑的情况会让用户出现误判，例如下面这个程序：

```
int main() {  
    int a, b, c, d, e;  
    //cout << "input a, b, c, d, e: "; //输出提示信息  
    cin >> a >> b >> c >> d >> e;  
    while( a + b + c + d + e < 100) {  
        a++;  
    }  
    cout << a << endl;  
    return 0;  
}
```

程序运行一开始就不作任何提示地要求用户输入多个数据，如果用户没有输入足够数量的数据（但是自以为已经输入足够多），则程序会一直在等待输入，这时用户会误以为程序出现了死循环。——由此可以看到，在让用户输入数据之前输出提示信息是很有必要的。

- 各种程序开发系统（或操作系统）都提供了某种方法，使编程者可以强制终止程序的运行。在 Dec-C++ 系统里，当程序正在运行时（出现死循环时程序也正在运行），**按快捷键Ctrl + Break就能终止其运行**。然后应该仔细检查程序，确定是否真是由于程序有错导致死循环，包括检查其中循环的终止条件等。请参考下一节和后面章节中有关排除程序动态错误的讨论。

第3章 变量和控制结构

- 3.1 语句、复合结构和顺序程序
- 3.2 变量——概念、定义和使用
- 3.3 数据输入
- 3.4 关系表达式与逻辑表达式
- 3.5 语句与控制结构
- 3.6 条件语句
- 3.7 循环语句
- 3.8 程序动态除错方法（一）

3.8 程序动态除错方法（一）

- 程序运行时，要仔细查看运行结果，利用专业知识进行分析判断是否正确。若有错误则需除错。
- 除错就是要排除自己编写程序时所犯的错误。
- 适合初学者的两种除错方法：
 1. 整理源代码缩进排版格式：（1）查看Dev-C++行号区的层次符号；（2）使用 AStyle 自动格式化源代码并查看是否有误；
 2. 在循环过程中输出查看变量的值。

- C和C++ 语言都是格式自由的语言，语言本身对源代码的排版格式并无要求。
- 但是在实践中，我们一定要严格地按一定的缩进排版格式编排程序代码。用**严格的缩进排版格式体现程序中的逻辑关系**，否则，阅读代码时就很难真正理解其中的逻辑关系，也比较容易产生误解。
- 从初学编程开始**就要特别注意养成良好的代码排版格式习惯**。经验告诉我们：**如果看不出程序里的错误，应先把程序的格式整理好**。

难以看出错误：

```
int n, sum = 0;
while (n < 100)
    sum += n;
n++;
```

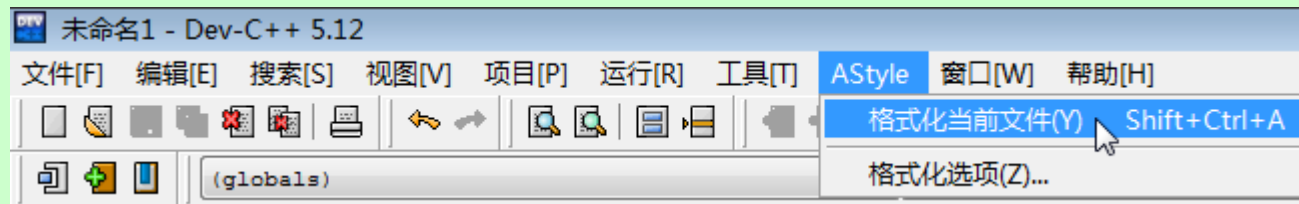
容易看出错误：

```
int n, sum = 0;
while (n < 100)
    sum += n;
n++;
```

- Dev-C++ 中有两个功能可以凸显源代码的逻辑结构，或帮助我们整理代码的缩进格式。
- 1、Dev-C++ 可以自动识别出源代码中的复合结构，并在编辑区左边的装订栏显示代码折叠按钮。

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int n, sum;
6     for(n=1, sum=0; n<=100; n++) {
7         sum = sum + n * n;
8         cout << "n= " << n << " sum= " << sum << endl;
9     }
10    cout << "sum= " << sum << endl;
11    return 0;
12 }
13
```

2、“AStyle”源代码自动格式化工具。对初学者很有用：使用这个工具对源代码重新调整缩进排版格式，注意观察调整之后的缩进排版格式与调整之前有什么区别，有可能发现程序中的问题。



输出在1-100之间能被7整除的数：

```
int main() {  
    int i;  
    for (i = 0; i < 100; i++);  
        if (i % 7 == 0)  
            cout << i << "\\t";  
    return 0;  
}
```

```
int main() {  
    int i;  
    for (i = 0; i < 100; i++);  
    if (i % 7 == 0)  
        cout << i << "\\t";  
    return 0;  
}                ? ? ?
```

- 有些同学弄不懂如何做出正确的缩进，那么请记住：在程序写完之后，就一定要运行一遍 AStyle！
- 经常使用它，就能自己越来越知道怎么写出正确的缩进排版了！

- 有读者可能说：既然AStyle能帮助调整排版格式，我编程序时就可以随便写，只要最后用AStyle调整就行了。

- **这是一种愚蠢的懒汉思想！**

按正确的格式编写程序，也是整理自己的思考成果并将其正确表现出来，AStyle不可能起这种作用，它只能帮编程者做辅助性的事后检验。

二、输出中间量的值

设法了解程序运行中发生的情况，一种常用的方法是：在代码中适当的位置加入输出语句，输出一些中间量的值。

例：计算从1到100区间的所有整数之和：

```
int main() {  
    int n, sum;  
    for (n = 1; n < 100; n++)  
        sum = sum + n;  
    cout << "sum= " << sum << endl;  
    return 0;  
}
```

sum= 2691806

在程序中的循环结构里加入一个输出语句，输出循环过程中变量 n 和 sum 的值。（需要把单条语句改为复合语句）：

```
int main() {  
    int n, sum;  
    for (n = 1; n < 100; n++ ) {  
        sum = sum + n;  
        cout << "n= " << n << " sum = " << sum << endl;  
    }  
    cout << "sum= " << sum << endl;  
    return 0;  
}
```

sum = 0

?

```
n= 1 sum =2686857  
n= 2 sum =2686859  
.....  
n= 98 sum =2691707  
n= 99 sum =2691806  
sum= 2691806
```

3.8.3 源代码的可读性

读者可能会在某些教材和资料上看到各种各样利用 C 和 C++ 语言的技巧而写出的奇形怪状、费涩难懂的程序（或片段），例如下面这样：

```
void main() {  
    int x = 3;  
    do  
        cout << (x -= 2) << " ";  
    while(!(--x));  
}
```

这种程序片段实际上有错误，违反语言的规范（把 main 函数写成 void 类型），而且在循环结构中故意写出费涩难懂的语句，要求读者去分析。

- 教科书上这样做是不合适的，可能把读者导向歧途。
- 源代码是专业人员的工作成果，其中沉淀了相关的思想和知识。不仅当前需要阅读和检查，将来还可能需要阅读、分析和修改。
- 因此，代码的可读性是极其重要的。逻辑清晰、简明易懂的源代码不仅更容易避免各种潜在的程序错误，也能方便编程人员之间互相交流、提高合作效率。
- 学习编程序，应该阅读并努力学会写作逻辑清晰、简明易懂的源代码。

```
int main( ) {  
    int x = 3;  
    do {  
        x -= 2;  
        cout << x << " ";  
        --x;  
    } while(x == 0);  
    return 0;  
}
```

- 提高代码的可读性可以节省代码阅读者的时间和精力（除错、扩展功能或是性能优化的前提条件是你要读懂这段代码），传达正确信息，避免误解。

下面列出的若干要素可供初学者参考：

- **变量名**采用助记、易理解的英语单词（或其缩写），也可以考虑用汉语拼音等；
- 采用人们编程实践中总结出来的**习惯用法**和**习惯写法**，如变量和常量名的统一构词方法，代码中的空格，长表达式（或语句）的换行和对齐，区分代码中不同部分的空行，等；
- 代码中的语句和控制结构（包括嵌套结构）应尽可能**清晰地表现计算的步骤和过程**，其中采用清晰的、常规的、易理解的处理方式；
- **恰到好处的注释**：
- 简单就是美，尽量**让每条语句只做一件事**